1

Information Systems Modeling

To provide a foundation for the discussions throughout this book, this chapter begins by defining what is actually meant by the term *information system*. The focus is on model-driven engineering of the software component of information systems. This chapter also introduces and describes the very notion of modeling. The chapter concludes with a brief discussion about software engineering processes, an important aspect of building successful information systems.

Definition of Information Systems

Information systems have played a key role in the history of computers and their use in everyday human activities. It is hard to imagine even a small company, institution, or organization that does not have a need for storing and using information of a different kind. We are all witnesses of the tremendous improvement of computer and communication technology, which support ever-increasing demands of human community for interchanging and utilizing information. It is not necessary to discuss the role and importance of information systems such as healthcare systems, enterprise systems, banking and financial systems, educational information systems, customer-support systems, governmental systems, and many other kinds of information systems (see Figure 1-1).

Trying to catch up with the importance and ever-emerging demand for improved functionality and performance of such systems, the hardware and software technology for their implementation seem to constantly stay behind. It has always been a question of how to improve the technology (especially that for software development) to meet the users' needs. Because this question defines the scope of this book, before trying to answer it, it is necessary to set up the context and define what information systems really are.

First and foremost, information systems are *systems*. A system is a set of elements organized to cooperate in order to accomplish a specific purpose. Elements of a system collaborate synergistically, in a manner that provides a behavior and quality bigger than the sum of its parts. It is likely that a

particular part of a complex information system (such as a piece of hardware or software) may be of no use if it is taken without the other parts. Only a complex interaction of such parts accomplishes the real purpose of the entire system.

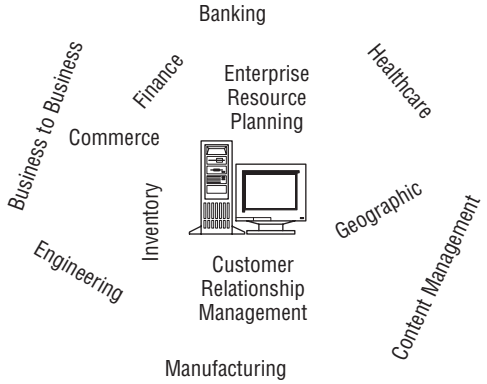


Figure 1-1: Information systems

Moreover, information systems are *computer-based* systems, meaning that their elements encompass hardware, communication equipment, various software subsystems, information, and users. The software of almost every information system is also a complex subsystem, comprised of communication software, the operating system, the database management system, different application modules, presentation modules, and so on. Because information systems generally assume storing and processing of large amounts of data, computer technology is irreplaceable for their implementation.

Next, information systems deal with *information*. Avoiding more complex and abstract definitions and focusing on computer-based information only, information can be understood as a piece of data that is structured, stored, transferred, processed, and presented in a proper manner, and at a right time, so that it has a certain meaning and accomplishes a specific purpose for its users. Therefore, information is not just a simple fact, but a piece of data that is shaped properly and provided timely to meet the specific user's needs.

For example, the fact that a certain patient has blood type A has no importance to the accountant using a hospital information system. To the accountant, it is much more important to present the total quantity of blood type A used for transfusions in a certain period. On the other hand, the same fact is key for a doctor who must organize the transfusion for that patient. However, if that fact relates to a patient that was hospitalized 50 years ago, it may be of no importance even to the doctor. If, however, the fact relates to a patient who just arrived in the Emergency Room, it has the crucial importance and represents the right information at the right time.

In summary, an information system is a computer-based system primarily dealing with pieces of data that are structured, stored, transferred, processed, and presented in a proper manner, and at the right time, so that they have a certain meaning and accomplish a specific purpose for the system's users.

For a more in-depth discussion of the variety of characteristics of information systems (characteristics that can more precisely describe the nature of information systems, as well as their engineering aspects), see Chapter 20 in the Supplement.

Section Summary

- An information system is a computer-based system primarily dealing with large amounts of data that are structured, stored, transferred, processed, and presented in a proper manner, and at a right time, so that they have a certain meaning and accomplish a specific purpose for the system's users.

Models and Modeling Paradigms, Languages, and Tools

Model-driven software engineering is a modern discipline that covers the building of software based on *models*, using modeling languages and tools. However, the notions and principles of this discipline have roots in other engineering approaches. This section further explores these topics.

Modeling

The engineering disciplines that are much more mature than software engineering (such as civil, mechanical, or electrical engineering) require that the designers of a new system build simplified representations of the system under construction in order to analyze the future system's characteristics before it is built, as well as to convey their design to the implementers. Such a simplified representation of the system under construction is called a *model* of the system, and the process of its creation is called *modeling*.

Models can be expressed in abstract terms, such as, for example, mathematical models, schemas, or blueprints. On the other hand, models can be physical, usually small copies made of plastic, wood, or other materials. Models allow the designers to experiment with the design and test its characteristics before the system is actually built. Such an approach reduces the risk of faulty construction of the real system with potentially disastrous or costly consequences.

There is no reason why software engineering would not follow these good practices of other engineering disciplines and exploit models and modeling in building complex software systems. One crucial difference is that in software engineering, the ''material'' that models are made of can be the same as the ultimate system — the model can be a formal and unambiguous specification that can be either directly interpreted by another software system, or transformed into a form that can be executed by hardware or interpreted by another software system. In that case, the model is *executable* and represents, when complete, the very system under construction.

A model is a simplified representation of the real world. It is built to provide a better understanding of a complex system being developed. Models of complex systems are built because such systems cannot be comprehended in their entirety. By modeling, four aims are achieved [Booch, 1999]:

- Models help to visualize a system as it is, or as designers want it to be (*visualization*).
- Models permit the designer to specify the structure and behavior of the system (*specification*).

- ❑ Models give templates that guide in constructing systems (*construction*).
- ❑ Models document the design decisions that have been made during development (*documentation*).

Basically, programs created in traditional programming languages may be treated as models because they specify the software system in a formal and executable way. However, programs created in traditional programming languages are not considered as being models, because of the following:

- ❑ Models basically deal with more abstract notions, while programs deal with subtle implementation details.
- ❑ Models may often be incomplete or imprecise, especially in early analysis phases, although this does not prevent them from being executable. Programs are ultimate artifacts of software development.
- ❑ Models are usually specified in visual (diagrammatic) notations, combined with textual (sequential) parts, while traditional programming languages predominantly presume textual (sequential) forms.

The software engineering discipline in which models are the central artifacts of development, which are used to construct the system, communicate design decisions, and generate other design artifacts, is called *model-driven engineering* (*MDE*), while the development of systems that exploit MDE is called *model-driven development* (*MDD*).

Modeling Languages

To build models, developers must know the vocabulary that can be used in modeling. The vocabulary available for modeling (that is, the set of concepts, along with the semantics of these concepts, their properties, relationships, and the syntax) form the definition of the *modeling language*. Using a certain modeling language, developers build sentences in that language, creating models that way. Therefore, a modeling language is the key tool available to developers for building models because there is no purpose in making models without understanding their meaning. Hence, a modeling language provides a common means for communication between developers.

To be successful, a modeling language must be carefully balanced to meet somewhat opposing demands.

On one hand, a modeling language must be simple enough to be easily understandable and usable by modelers. If the modeling language were too complex to be comprehensible, it would not be widely accepted by the software community. Additionally, it is very desirable that users who pose the requirements also understand the modeling language, at least the part that is used in specification of requirements. This property may significantly reduce the risk of misunderstandings between users and modelers during requirements specification.

On the other hand, a modeling language should not be too simple or too specific. If the modeling language were not general enough, it could not cover all situations that could arise in the real world.

In addition, a modeling language must be abstract enough to be conceptually close to the problem domain. A modeling language that is not abstract enough suffers from a large conceptual distance

from the problem domain. In that case, the development requires a big mental effort because the modelers must take the conceptual mapping from the problem domain into the model. In other words, the language must be *expressive* enough.

Expressiveness is a descriptive property of a language that measures the ''quantity'' of meaning of certain language concepts. An expressive language consists of concepts that have rich semantics and, thus, have plentiful manifestation at the system's execution time. Models in an expressive language may be concise, and yet provide lots of run-time manifestations that correspond to the posed requirements. Expressiveness is, in other words, a measure of conciseness with which a particular logical design may be expressed in a certain language. Put more simply, it measures how many (or few) words you must say in the given language in order to express some logical design or intention.

Therefore, the modeling effort and development cost is directly affected by expressiveness. If the language is expressive enough, models consist of smaller sentences, and developers make less effort to specify systems under construction, and vice versa.

Additionally, a modeling language should allow informal, incomplete, or inconsistent models because the process of modeling in the early phases of requirements specification and system conception often assumes such models. During the early phases of requirements engineering and system conception, system analysts and designers have vague and sometimes incorrect visions of the systems being specified and constructed. Therefore, the analysts should be able to make sketches in the modeling language, in a way that allows their later refinement. If the modeling language does not allow such imprecise modeling, it could not be used in the early requirements specification and conceptualization phase. However, it is very useful if the same language is used in the entire process of system development, but not just during requirements specification or just during design.

On the other hand, a modeling language should be simple and primitive enough in order to have precise semantics that allow transformation of models into an implementation form. Highly abstract models are usually transformed into lower-level forms that can be either interpreted by other software systems, or further transformed into even lower-level forms, or ultimately executed by hardware (which also interprets the binary code in some way).

For example, source code in a traditional textual programming language such as C++, Java, or C# is transformed (compiled) by other software systems (compilers) into either a binary executable program (for C++), which is executed by hardware, or into an intermediate form (as in Java or C#), which is interpreted by other software systems (virtual machines), to provide the running application. Similarly, a model made in a visual modeling language, such as the Unified Modeling Language (UML), which is a focus of this book, may be transformed into the target programming language code (for example, C++, Java, or C#).

The described transformation into a lower-level form can be performed manually, semi-automatically, or completely automatically, using the corresponding transformer. It is very useful if a highly abstract model of a system, specified in a modeling language, can be transformed completely automatically (in one or several steps) into a form that can be executed or interpreted in a way that provides the running application, following the semantics of the initial model. In that case, the modeling language can be

treated as executable, although it is not directly executed by hardware. This is because the model implies an automatically achievable form that can be ultimately executed by hardware.

In this chain of transformations and intermediate models (for example, source code, intermediate code, executable code, and so on), the form that can be interpreted by another software system to result in the running application (such as Java byte code or SQL statements), or that can be automatically compiled into an executable binary code (such as source code in C++), is referred to as the *implementation form*, and the language it is made in is referred to as the *implementation language*.

In order to be executable, the modeling language must have precise and completely formal semantics, and the model that should be executed (or, more precisely, automatically transformed into an implementation form), must be formal, unambiguous, and consistent. These are the most important characteristics of the modeling languages that claim to be useable and useful for modeling of any complex software system in general, and information systems in particular.

The concrete notation of modeling and implementation languages can be textual or visual. Textual languages allow you to create inherently *sequential* models. Although textual models (that is, programs or scripts or sentences written in textual languages) may be visually perceived by the developer in two dimensions, they are inherently sequential because the transformers parse them as sequences of characters.

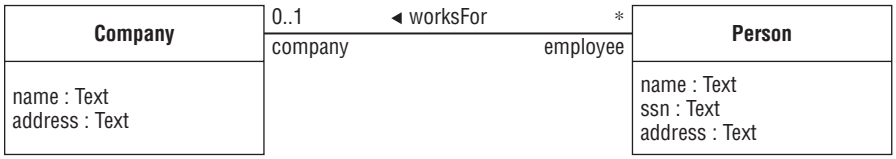
Conversely, visual languages assume making models in two or even three dimensions, most often using diagrammatic notations combined with textual fragments. It is usually the case that a pictorial, diagrammatic notation is much more descriptive, intuitive, and easier to perceive than the equivalent sequential (textual) form. This is why modeling languages with diagrammatic notations are more popular for abstract modeling.

However, it is not always true that visual languages are more usable than textual ones. Some intentions can be much more effectively expressed in a textual form than in a visual form. For example, to specify a simple loop or an expression such as a regular expression or a simple addition of two operands, a textual form may be much more convenient and concise than an equivalent graphical notation.

In short, both textual and diagrammatic languages have their advantages and drawbacks. Usually, a combination of both is most efficient in modeling.

Figure 1-2 illustrates the comparison of abstract versus low-level modeling, and visual versus textual modeling languages. A small piece of a model in a highly abstract, visual language is shown in Figure 1-2a, whereas the equivalent model in a low-level, textual language is shown in Figure 1-2b. Both examples model the same simple fact from the problem domain that a person (described with name, social security number, and address) may work for at most one company (described with name and address), and a company may employ many persons.

The example shown in Figure 1-3 illustrates a yet more dramatic difference between the abstract visual model in Figure 1-3a, and its semantic equivalent in a lower-level, textual programming language in Figure 1-3b. The figure shows the lifecycle model for a course enrollment in an imaginary educational institution. The diagram in Figure 1-3a depicts how a student's application for a course is processed until it gets either rejected or accepted, or suspended (and later resumed) or canceled.



(a)

```
class Company {
public:
  String name;
  String address;
  Collection<Person*> employee;
};
```

```
class Person {
public:
  String name;
  String ssn;
  String address;
  Company* company;
};
```

(b)

Figure 1-2: Abstract vs. low-level modeling and visual vs. textual modeling languages. (a) A model in a highly abstract, visual language. (b) A model in a low-level, textual language. Both examples model the same simple fact from the problem domain that a person (described with name, social security number, and address) may work for at most one company (described with name and address), and a company may employ many persons.

Modeling Tools

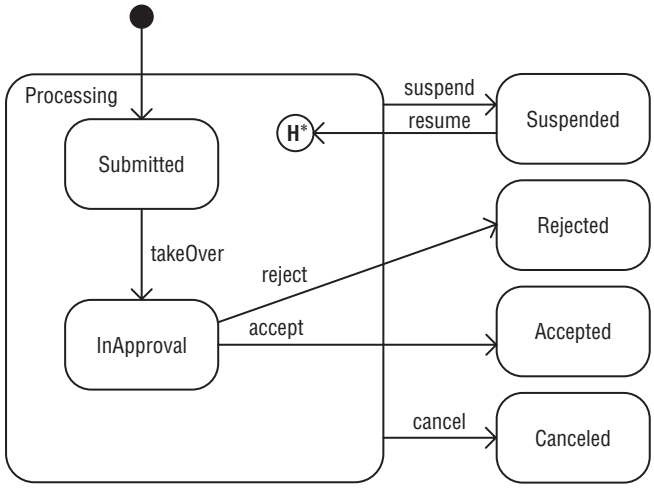
As in any other engineering discipline, tools are key factors to successful production. In the context of modeling, different kinds of software systems are used as modeling tools. The system that is used as an environment for creating and editing models (and that is responsible for their specification, visualization, and consistency checking) is called the *model editor*. The tools that transform the models into lower-level forms are generally called *transformers* or *translators*, or specifically *generators* or *compilers*, depending on their target domain.

For example, a tool that transforms a higher-level, abstract, and visual model into a textual source code in the implementation language (for example, C++, Java, or C#) is often referred to as a *code generator*, although it is sometimes called a model compiler.

Finally, the tool that transforms a program written in a textual language (such as C++, Java, or C#) into the executable (binary) code, or interpretable (intermediate) code, is often called the *compiler*. Many other kinds of modeling tools are also used for specific tasks in development, such as model analyzers, model comparators.

Modeling Paradigms

There are modeling languages that are particularly appropriate for (or the same) problem domains, such as those that analyze or compare models, and others. These languages, based on the same set of fundamental concepts, form a *modeling paradigm*. The languages of the same paradigm may differ in more or less detail or variations.



(a)

```
switch (state) {
  case Submitted:
    switch (event) {
      case takeOver: state = InApproval; break;
      case suspend:  prevState = state; state = Suspended; break;
      case cancel:   state = Canceled; break;
    };
    break;
  case InApproval:
    switch (event) {
      case reject: state = Rejected; break;
      case accept: state = Accepted; break;
      case suspend: prevState = state; state = Suspended; break;
      case cancel:  state = Canceled; break;
    };
    break;
  case Suspended:
    if (event==resume) state = prevState; break;
}
```

(b)

Figure 1-3: Abstract vs. low-level modeling and visual vs. textual modeling languages. (a) A model in a highly abstract, visual language. (b) A model in a low-level, textual language. Both examples model the lifecycle of an application for a course.

For example, the relational modeling paradigm is based on the fundamental concept of mathematical relation, which is often represented by a table with columns (or fields, properties, attributes) and rows (records or tuples). Although relational database management systems (RDBMSs) often support somewhat different languages, varying in other concepts built upon the basic relational concepts, they are still based on the same paradigm. For additional information about the relational paradigm and DBMSs, see Chapter 22 in the Supplement.

Similarly, the procedural programming paradigm assumes some common concepts such as data type, variable, expression, statement, loop, subprogram (procedure or function), argument (formal and actual),

subprogram invocation, recursion, and so on. Many programming languages that fall in this category (for example, Algol, Pascal, C, Fortran, and so on) support most of these concepts in very similar ways, but differ in many other less relevant details.

This book is devoted to a standard modeling language for general software modeling, called the *Unified Modeling Language* (UML). This language supports the object paradigm, significantly different from the relational paradigm, for example. The basic concepts of the object paradigm are summarized in Chapter 24 of the Supplement, but will also be explained from the beginning in Part II.

Section Summary

- A *model* is an abstract, simplified, or incomplete description of the system being constructed.
- The language used for creating models is called the *modeling language*. It encompasses concepts used for creating models of systems, their semantics, properties, and relationships, along with the syntax for creating correct models.
- A modeling language should be simple enough to be easily comprehensible and usable by modelers. On the other hand, the language should not be too simple or too specific.
- A modeling language must be expressive and abstract enough to be conceptually close to the problem domain. However, the language should be simple and primitive enough to have precise semantics that allow transformation of models into implementation forms.
- If the concepts of a modeling language have formal semantics that enable models to be transformed completely automatically into forms that can be executed or interpreted in a way that results in running applications, the language is called *executable*.
- The form that can be interpreted by another software system that provides the running application, or can be automatically compiled into an executable binary code, is referred to as the *implementation form*, and the language it is made in is called the *implementation language*.
- The concrete syntax of modeling languages can be visual (diagrammatic) or textual (sequential).
- The software engineering discipline in which models are the central artifacts of development (which are used to construct the system, communicate design decisions, and generate other design artifacts) is called *model-driven engineering* (MDE), while the development of systems that exploits MDE is called *model-driven development* (MDD).
- The tool that is used for creating and editing models, responsible for their specification, visualization, and consistency checking, is called the *model editor*.
- The set of languages based on the same set of fundamental concepts form a *modeling paradigm*.

Processes and Methods

A modeling language is just a vehicle for modeling. It only specifies the concepts that can be used in modeling, as well as their semantics and the rules for forming correct models. However, the language itself does not give any direction *how* to specify requirements, create models, or develop a system. In other words, the *process* of development is not defined by the language itself.

In general, a process is a set of partially ordered steps intended to reach a goal. In software engineering, the goal is to efficiently and predictably deliver a software product that meets the users' needs [Booch, 1999].

In software production, at least two types of processes can be distinguished, depending on their scale and focus.

First, there is a higher-level *project management* process that deals with planning, organizing, supervising, and controlling phases and iterations during the entire project, interacting with the customers, managing resources, as well as defining milestones and artifacts of development phases. It is mostly independent of which modeling language or paradigm is exploited.

On the other hand, there is a lower-level *design process* that deals with how the modeling language and other technology is used and applied in different situations, as well as other things such as selecting or inventing suitable design patterns for given design situations, refactoring techniques, ways of using tools, and the like. To successfully build systems, designers must follow a proper design process that, in conjunction with the used modeling language, forms the *modeling* (or *design*) *method*.

For more information about the characteristics of project management process models, see Chapter 21 in the Supplement section. Part IV describes some elements of the design process proposed for the modeling method described in this book.

At this point, let's just briefly summarize some of the key properties of the proposed design method.

First, the proposed design method is based on the object paradigm in all its phases and all its artifacts. Second, it is oriented to production of abstract models instead of paper documents, as the principal artifacts of software development.

In addition, it encourages reuse of different artifacts and relies on a formal, executable modeling language as much as possible, and as early as possible. Such orientation can improve software production dramatically. Namely, if an executable model of the system being constructed can be obtained early enough, as soon as the requirements are captured, but without exhausting conceptual mappings, the ultimate implementation can then be reached automatically and rapidly. As already stated, the basic precondition for this is a highly abstract, but formal and executable modeling language, as well as a design process that supports its use. This is also one of the key focuses of this book.

Section Summary

- A process is a set of partially ordered steps intended to efficiently and predictably deliver a software product that meets users' needs.
- The *project management* process deals with planning, organizing, supervising, and controlling phases and iterations during the entire project, interacting with the customers, managing resources, as well as defining milestones and artifacts of development phases.
- The *design process* deals with how the modeling language is used and applied in different situations, as well as other things such as selecting or inventing suitable design patterns for given design situations, refactoring techniques, ways of using tools, and the like.
- The design method proposed in this book relies on using formal and executable models, as well as producing such models as early as possible.